Remote Subpaging Across a Fast Network *

Manjunath Bangalore and Anand Sivasubramaniam

Department of Computer Science & Engineering
The Pennsylvania State University
University Park, PA 16802.
Phone: (814) 865-1406
{bangalor,anand}@cse.psu.edu

Abstract. While improvements in semiconductor technology have made it possible to accommodate a large physical memory in today's machines, the need for supporting an even larger virtual address space continues unabated. Improvements in disk access times have however lagged improvements in both processor and memory speeds. Recent advances in networking technology has made it possible to go out on the network and access the physical memory on other machines at a cost lower than accessing the local disk. This paper describes a system implemented for such a remote paging environment. This system allows us to use a fine grain (a subpage) data transfer unit for remote memory paging and to employ different algorithms for determining when and how to transfer these units. The novelty of our implementation is that all the policy decisions about the subpage size and the subpaging algorithm are made at the user level, thus letting applications choose their own set of parameters. Performance results indicate that applications can benefit significantly from this flexibility.

1 Introduction

Applications have traditionally demanded a larger address space than available physical memory in machines. The operating system virtual memory management via paging has hidden this limitation from the programmer by transparently transferring pages between physical memory and the swap device. Until recently, the local disk has been the obvious choice for the swap device because any non-local repository has had to use a relatively slow network.

However, recent advances in networking has given us high-bandwidth, switched networks such as ATM [5] and Myrinet [2]. Further, low-latency messaging layers such as Active Messages [6] and Fast Messages [13] can exploit almost the entire promised capabilities of these networks by drastically reducing software costs. As a result, it is now a lot less expensive to go out on the network and

* This research is supported in part by a NSF Career Award MIP-9701475, and equipment grants from NSF and IBM.

D. K. Panda and C. B. Stunkel (Eds.): Network-Based Parallel Computing, CANPC 98
LNCS 1362, pp. 74–87, 1998.

access the physical memory of another machine than accessing the local disk. Figure 1 illustrates this point by comparing the transfer times of different data sizes for Myrinet using Fast Messages and a recent disk (a Western Digital Enterprise 9.1 GB Ultra SCSI Harddrive).

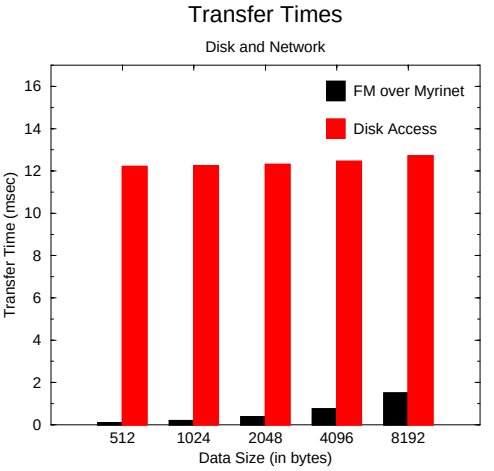

Fig. 1. Comparison of Network and Disk Accesses

A recent study [1] observes that at any particular instant, a large number of machines on a network are idle. For instance, in a network of 50 workstations, around 30 were found to be idle at any given time in this study. An application on a machine could thus benefit from the idle physical memory on the other machines. Remote paging can help better utilize and manage physical memory on a global scale across all machines.

To harness and manage the RAM across the machines on the network, there are three important issues that need to be addressed. First, we need to quantify the size of the transfer unit (called the subpage) across the network. Second, we need to implement an access control mechanism that can efficiently handle this transfer unit. Finally, we need to investigate different algorithms for fetching subpages from remote memory.

The size of the transfer unit for a paging system depends on the application memory reference pattern, the size of the physical memory, and the difference between the access times of memory and the swap device. When we move from traditional to remote paging, the only difference is the access time for the swap device. Since paging using network RAM is expected to be more efficient than using a disk (Figure 1), the transfer unit for remote paging should be smaller than a traditional page.

Hardware access control is essential to detect accesses to unmapped pages and writes to read-only pages. The MMU hardware is usually tailored for a specific page size and is not always controllable in software (and may not allow variable page sizes). Our goal is to provide remote paging mechanisms on

off-the-shelf workstations and networking hardware, so that they can be readily used in commonplace platforms. Hence, we do not want custom MMU hardware for implementing access control at a finer granularity. In this exercise, we use the Wisconsin Blizzard-E [7] drivers on commodity hardware, to implement access control for subpages on SPARCstation 20 platforms. These drivers modify Solaris' cacheability assumptions about memory and devices, and provide ioctl calls to the user programs to implement fine grain access control. We have chosen this approach over the all-software Shasta [14] approach to avoid incurring any overheads when the accessed memory location is resident locally (which is the more frequent case).

If the hardware determined page size cannot be altered, physical memory allocation has to be still done on the traditional page basis and not on a subpage basis (or else, adjacent virtual pages may not necessarily be physically adjacent). Consequently, we could have situations where one or more subpages within a page are present while the remaining reside on a remote node. It would thus be interesting to study different ways of prefetching these remaining subpages, and compare these schemes to a purely demand based subpaging scheme.

Other studies related to remote paging [12, 3, 10, 11, 4, 9, 17] have considered a spectrum of issues from server loads, to replacement schemes and global memory management issues. It should be noted that our system can be used in conjunction with any of these ideas. The closest study to ours is by Jamrozik et al. [8] where the benefit of subpages to improve performance is shown via trace-driven simulation. They have validated their results with a prototype implementation on the DEC Alpha connected to an ATM network. Fine-grain access in their case is achieved by modifying the PAL code of the memory subsystem. However, their remote paging system is not implemented at the user-level.

This paper presents an implementation of a remote subpaging system addressing some of the above mentioned issues. This system has been implemented on a SPARCstation 20 platform that has a page size of 4K bytes. The user-level Fast Messages [13] layer over Myrinet has been employed for communication. The novelty of this system is that it is customizable to an application's needs in terms of the transfer unit and the remote subpaging algorithms since most of it is implemented as a user-level library. Several applications could thus co-exist on a machine, each with its own choice of subpage size and remote subpaging algorithm.

The design of the remote subpaging system is discussed in Section 2 and the performance results are presented in Section 3. Finally, we present concluding remarks and ongoing research in Section 4.

2 Remote Paging System Design

Towards our overall goal of developing an efficient remote paging system, there are three important issues that need to be answered :

– What is the ideal unit of data transfer *(subpage)* between the workstations on the network? Can we simultaneously support multiple subpage sizes so that each application can choose its own subpage size?
– How do we efficiently implement access control for this transfer unit?
– What are the different remote subpaging algorithms that should be supported? How do these algorithms compare? Can we offer the applications the flexibility to tailor the subpaging algorithms to suit their needs?

To investigate these issues, we have developed a remote subpaging system on SPARCstation 20 platforms connected by Myrinet employing a user-level, low-latency, high-bandwidth messaging layer (Fast Messages [13]). In this paper, we specifically concentrate on issues in implementing the remote paging system at the machine where the application is executing, which we shall call the client, and use a simple server to store and retrieve these pages at the remote machine. There are several research issues in the design of the server but these are beyond the scope of this paper.

There are two basic modules in our implementation of the remote paging system at the client. The first is a *system of loadable device drivers* that executes in the Solaris kernel to provide mechanisms that are needed to implement fine grain access control. The second is a user-level library, linked with the user program, which uses the OS mechanisms to implement different remote subpaging algorithms.

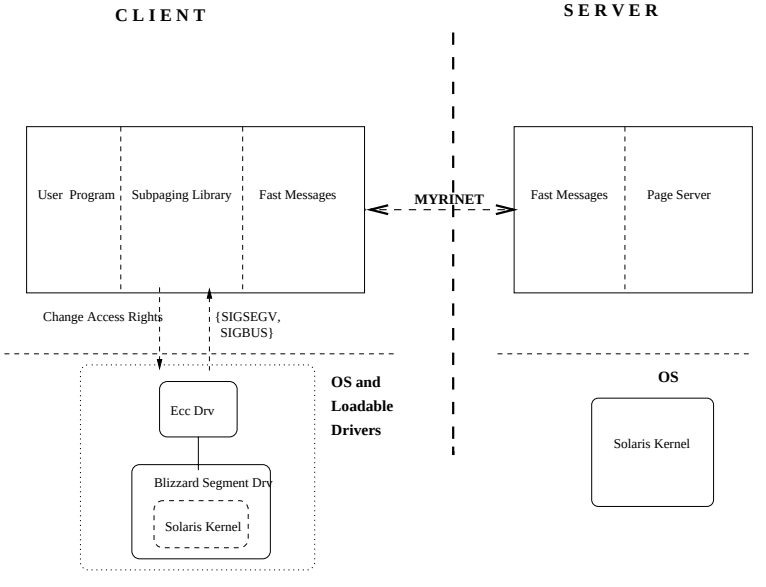

Fig. 2. The Remote Paging System

2.1 The Fine-grain Access Subsystem (Loadable Device Drivers)

Since accesses to remote memory across the network is less expensive than accesses to the local disk (which is the main motivation for this work), the unit of data transfer for remote paging should be smaller (*subpage*) than the normal page. However, not all traditional hardware for virtual memory can be programmed for alternate transfer units. Hence, for finer-grain (smaller than a page) access control, we have to resort to other means. In this exercise, we have used the loadable driver subsystem from the Wisconsin Blizzard-E [7] that provides fine-grain access control for the SPARCstation 20 platform running the Solaris 2.4 operating system. These drivers deliberately force uncorrectable errors in the memory's error correcting code(ECC) and use virtual memory aliases in addition to implement fine-grain access control. They allow a user-level process to set specific parts of a page (subpage) to VALID/INVALID, and a user-level SIGBUS handler is invoked on access violations. Also, a SIGSEGV handler is invoked for accesses to an unmapped page. The fine-grain access subsystem realizes the INVALID state by forcing uncorrectable errors in the memory's error correcting code (ECC) while guaranteeing no loss of reliability. Resetting the ECC bits is done with uncached double word stores using a second uncached mapping on the same pages.

2.2 The Subpaging Library

Our system tries to adhere to the well-known operating system philosophy of separating mechanisms from policy. The kernel component of our system (the loadable drivers) provides only the mechanisms for fine-grain access control. The rest of the remote paging system is implemented as a user-level library. This separation also helps us provide a menu of remote subpaging algorithms and a choice of subpage sizes that each application can choose from based on its expected behavior. Protection violations at page level (first accesses to unmapped pages) are detected by the VM hardware and passed to the user-level handler through the UNIX SIGSEGV signal handling interface and access violations detected by the driver are passed on to the user-level library via the SIGBUS (subpage fault) signals. The library employs the driver to change access rights of subpages when needed. The reader should note that we are not violating any of the traditional memory protection requirements with this design since the user-level library can only manipulate its own pages (and not belonging to anyone else).

2.3 Implementation Details

For a better understanding of our system, let us walk through the sequence of actions in a typical execution. Memory (to be remote paged) is allocated via our user library routine which in turn calls the kernel drivers to initialize the region of memory to an "UNMAPPED" state. An access to any part of this "UNMAPPED" memory by the application would result in a fault (page fault) which is passed on to the user library via a SIGSEGV signal. If space is available

for this referenced page in local memory, then a physical page is allocated with a default protection of "INVALID" on the entire page using the kernel driver. The cost of this operation is directly proportional to the page size. The state for the faulted subpage alone is then set to the "VALID" state. Our system restricts subpage sizes to powers of 2.

At this stage, even though the physical page frame has been allocated on the client, the data itself resides on the remote machine. The subpage referred to by the user program access has to be fetched from the server by the SIGSEGV handler before the application can proceed. A subpage request is sent to the server to which the server replies with the necessary subpage. This subpage is then copied to the appropriate part of the allocated physical frame at the client, and control returns to the application program.

When there is no space left in the client physical memory, a victim has to be evicted and sent to the server, and that page is set to "UNMAPPED" at the client. We could use any of the well-known page replacement algorithms to choose the victim. Since our focus here is on subpaging issues, we have chosen a simple FIFO scheme for page replacement, and fancier schemes such as LRU can be substituted without changing the design of the subpaging system. As long as the application keeps referencing the "VALID" subpage in a page, it will not incur further access violations and overheads. The moment it references another ("INVALID") subpage within that page, there is an access violation (note that this is not a page fault since the page has been mapped in), caught by the driver through a memory error (via ECC bits), and passed on to the user-level library by a SIGBUS signal. The user-level SIGBUS handler may have to send a request for the subpage to the server if that subpage has not been already received by the client (which is possible in certain subpaging algorithms to be discussed shortly). The state for that subpage is set to "VALID" using an ioctl call to the kernel driver. When the subpage is received, it is copied to the appropriate part of the physical page. Control then returns to the application which can proceed with accesses to that subpage without overheads.

As we mentioned earlier, the scope of this paper is limited to the client side issues. There are interesting ideas to be explored on the server side which we plan to investigate in the future. Currently, the server side software simply pins the pages that it controls in its physical memory and serves the client requests (*keep-page* and *get-page*) one after another.

2.4 Subpaging Schemes

Once a physical page frame is allocated and mapped in at the client, there are different schemes by which one may fetch the different subpages within that page from the server. In this paper, we examine three such schemes that are described below.

Demand Subpaging: In this scheme, each subpage of a page is fetched one at a time, and only when that subpage is referenced. Each SIGSEGV and SIGBUS handler invocation has to necessarily fetch the referenced subpage from the

server. The advantage of this scheme is that it does not transfer any more subpages than needed. The disadvantage is that there is no overlap of data transfer while the CPU is executing the application program. The application has to stall while the request is sent on the network to the server and the server sends back that subpage over the network.

Eager Subpaging: This scheme attempts to overlap data transfer with useful work execution. The SIGSEGV handler works the same way as before on the client side. On the server side however, the server sends the requested subpage in one message. Then, in another message, it automatically takes the remaining subpages of this page and sends them to the client without requiring the client to explicitly ask for them. The SIGBUS handler (at least one subpage request in that page has already been processed) on the client simply polls the network for the requested subpage since the server would automatically send it. It may happen that the user-level library at the client picks up subpages (other than what it is waiting for) and sets those subpages to "VALID". This is possible when the application is referencing multiple pages in a short time span.

The advantage of the Eager scheme is that it could potentially hide data transfer latencies if the gap between the first reference to a page and the first reference to another subpage within that page is sufficiently large. The disadvantage is that the server may be sending more subpages to the client than what is actually needed. Further, even though the SIGBUS handler is waiting for only one subpage it may have to wait longer since the server is sending all the subpages in one shot. It is possible to "stream" the subpages from the server to the client to fix this problem, but we have not explored this improvement in this paper.

Forward Sweep Subpaging: This is a variation of the Eager scheme where the faulted subpage is fetched (on a SIGSEGV), the application continues execution and the server asynchronously sends only the remainder of the page that is ahead/forward (in terms of the address) of the faulted subpage in a single unit. If the application accesses the portion of the page that is behind the faulted subpage, a SIGBUS is incurred. The motivation for this approach draws from a finding in a recent study [8] that in a subpaging system, most of the accesses to a page are in the region that is ahead of the faulted subpage with 70% of them being on the immediately next subpage in the forward direction. This scheme is expected to perform even better than the Eager scheme under such situations since the polling time in a SIGBUS handler is likely to become lower because of the smaller number of subpages being sent.

3 Performance Results

Having, described our remote paging system, we now evaluate the performance of this system. First, we examine the time spent in the different components

of the system using microbenchmarks. Next, we evaluate its performance using traces from three of the Spec92 [15] benchmarks with different subpage sizes and subpaging schemes. Finally, we look at how an actual Quicksort application performs with different subpage sizes and subpaging algorithms. We have evaluated this system over two SPARCstation 20s with Hypersparc processors connected by Myrinet through an 8-port switch.

3.1 Microbenchmarks

Subpage Size (in bytes)	Allocate New Page	Set Subpage Valid	Remote Subpage Fetch
2048	1930	150	230
1024	1930	125	167
512	1930	99	119
256	1930	95	111

Table 1. SIGSEGV handling cost in microsecs

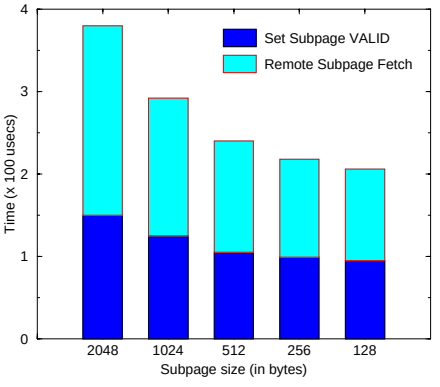

Fig. 3. SIGBUS handling cost

Let us examine where time is spent in handling a typical fault in the remote subpaging system. We have two types of faults namely the SIGSEGV which is incurred on a first-time access to an unmapped page (page fault), and the SIGBUS which is caused on future accesses to a partially resident page (subpage fault). We have measured the time spent in these fault handling routines with a simple microbenchmark. The different operations performed on a SIGSEGV

violation are declaring a new page, setting the faulted subpage valid, and fetching the faulted subpage from remote memory. The cost of these operations for different subpage sizes is shown in Table 3.1.

We can see that allocating a new page is a costly operation and takes nearly 2 milliseconds. The cost for setting the subpage valid and the cost for fetching the subpage from remote memory changes linearly with the subpage size. The SIGBUS handler has to set a subpage valid and has to fetch the faulted subpage from remote memory. The cost of these operations is shown in Figure 3.

These operations may seem expensive (even though they are much cheaper than the disk accesses), but we are currently in the process of switching to a fast trap interface on Solaris to transfer signals efficiently (around 5 microsecs) to the user handler.

3.2 Experiments with Traces

In this subsection, we evaluate the performance of the subpaging system by using three traces from the Spec92 benchmarks suite [15] namely gcc, compress, and espresso, which exhibit different memory access patterns. We have used data traces with at least 150,000 references and perform reads and writes depending on the nature of the reference. There is no computation (gap) between these memory references.

In the following discussion, we present results from running these traces on our system with varying subpage sizes and subpaging algorithms. We have also obtained various statistics about the number of SIGSEGV and SIGBUS invocations to explain the behavior of these applications, though they are not given explicitly in this paper. Also, since our focus in this exercise is on the subpaging issues, we use a simple variation of FIFO for page replacement (it is FIFO with higher priority given for partially filled pages). In all the exercises, we have set the amount of physical memory available at the client node to half the virtual memory requirement of the application.

The performance results for the gcc trace with 200,000 references, the compress trace with 200,000 references and the espresso trace with 150,000 references are shown in Figure 4

The advantages of a smaller transfer unit (than a page) for gcc is brought out in the 2K demand and forward sweep subpaging schemes where there is a 13% improvement in performance over the full page size. *Note that the bars for 4096 bytes in these graphs correspond to the results for access control and remote paging at the full page granularity.* From 1K bytes onwards, with decreasing subpage size, the performance suffers due to an increased number of SIGBUS faults. This gets worse with smaller subpage size. While the eager scheme does better, the downside of eagerly transferring the rest of the page in one shot (after the first reference) seems to cause the CPU to stall more than it should. Sending a smaller chunk in the forward scheme helps. The advantages in lowering the number of SIGBUS faults with the forward sweep scheme (compared to the demand based scheme) and not letting the CPU stall too long on a second

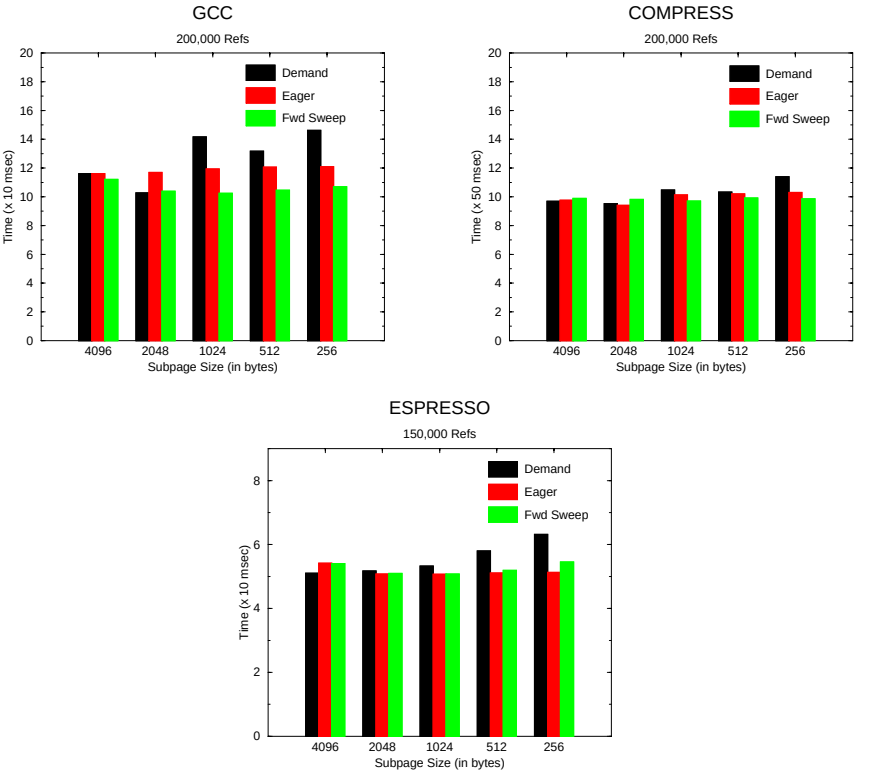

Fig. 4. Performance Results for Traces

subpage reference helps gcc perform the best for this scheme at a subpage size of 1K bytes.

In compress, as one would expect of this application, there is a substantial spatial locality of reference, even at a subpage granularity. Hence, there is not a significant impact from the subpaging algorithm and the subpage size (only a marginal difference) compared to gcc. Still, the eager and forward sweep schemes do slightly better than the demand based scheme since nearly all the subpages within a page are referenced eventually in this application. Here again we find a subpage size of around 1K to give the best performance.

In espresso, the number of page faults (SIGSEGV) itself is fairly low. Once the page is brought in, the references are uniformly distributed spatially and temporally through the page. This makes the eager scheme perform slightly better than the forward sweep scheme. Also, since the number of SIGSEGV violations is low, the higher cost paid by these two schemes in the initial references to a page is not a significant overhead compared to the numerous SIGBUS faults that the demand based scheme experiences.

3.3 Experiments with an application

One drawback of using traces to examine the behavior of our system as in the previous subsection is that it does not give a realistic picture of what happens in a real execution. Usually, a program does not just make memory references. There is a certain amount of computation performed between these references. While this does not affect the behavior of the demand based subpaging scheme (which does not overlap any communication with possible computation), not modeling this computation is unfair to the other two schemes.

To study the system with an actual application, we have used a Quicksort program to sort 64K integers and we have studied the impact of subpage size and subpaging algorithm on this program. Again, we have set an artificial limit on the available physical memory of the client equal to half the virtual memory size requirements of the program.

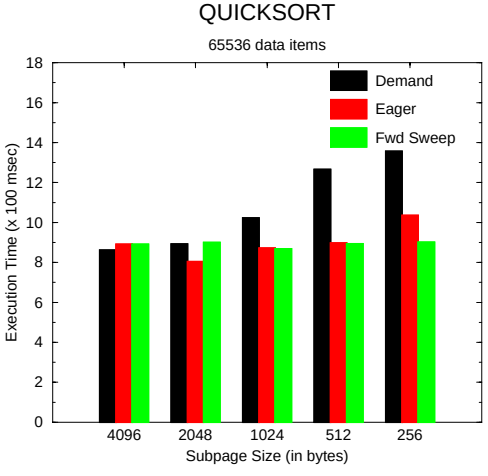

Fig. 5. Total Execution Time for Quicksort

All schemes are seen to perform best (see Figure 5) at subpage sizes of 2K or 1K bytes and the best performance is seen at 2K byte subpage size in the eager scheme. The demand scheme worsens when we go for any transfer unit smaller than a page. This is because the quicksort algorithm eventually accesses every part of a page. Since there is a reasonable amount of computation to be overlapped with fetching the remaining subpages of a page, the other two schemes perform much better. For subpage sizes shorter than 1K bytes, the forward sweep scheme performs better than the eager scheme and the performance is reversed for the larger subpage sizes. The best performance of the lot is for the eager scheme with a subpage size of 2K bytes which is around 11% better than the performance at full page size. While it may appear to the reader that a 11%

saving in execution time is not very significant, we should note that page sizes on machines are likely to get larger (to improve TLB coverage as the physical memory on the machines keeps growing). Hence, transfer units smaller than a page over the network will become even more important with this trend.

4 Concluding Remarks and Future Work

In this paper, we have presented a remote subpaging system that uses recent innovations in networking technology and communication software to provide an alternate and more efficient repository between the physical memory and the disk in the storage hierarchy. We have implemented this system on a SPARCstation 20 platform, connected by Myrinet, using a low-latency user-level messaging layer. The system allows us to use a fine grain (a subpage) data transfer unit for remote memory paging and to employ different algorithms for determining when and how to transfer these units. The novelty of this system is that all the policy decisions about the subpage size and the subpaging algorithm are made at the user level. As our performance results clearly indicate, each application could potentially benefit from a different subpage size and subpaging algorithm. Our system makes this customization possible without interfering with the protection requirements by implementing these policy decisions at the user level.

For most applications, a subpage size of 2K or 1K bytes is seen to give the best performance. In some cases, with these subpage sizes, there was close to a 15% savings in execution time over the full page size on the SPARCstation 20 (4096 bytes). However, the reader should note that page sizes on machines are likely to get larger to improve TLB coverage [16] as the physical memory on the machines keeps growing. This trend is apparent even on the newer SUN UltraSPARC platforms where the page sizes have grown to 8192 bytes. Hence, transfer units smaller than a page over the network will become even more important with this trend.

There are several interesting directions for future work that are currently being investigated as identified below:

- We are trying to use a faster way of processing signals as provided by the loadable device driver under Solaris. This scheme tries to minimize the number of protection boundaries that need to be crossed for every signal. This mechanism can be used for processing SIGBUS faults and would boost the performance of the remote subpaging system.
- We are investigating other subpaging algorithms wherein the different subpages could be streamed to the client one after another without being sent in one big chunk. A variation on the same lines is to get the subpages in the order that they are likely to be accessed.
- Another possibility is to perform subpaging in the kernel space (possibly in the underlying segment drivers that implement fine grain access control) to reduce switching overheads between the kernel and the user space while processing a fault. A comparative study between the performance benefits

of subpaging at kernel-level and the flexibility of subpaging at user-level can be conducted.
- It is interesting to find out what tools the user would need to choose an appropriate subpage size and subpaging algorithm for an application, to benefit from this system.
- The current system is being ported to UltraSPARC architectures running a newer Solaris release.
- There are several interesting research issues to be addressed for the design of the server.

We are expanding our system to provide a custom malloc interface that can manage the dynamic data requirements of applications to use fine grain access control. This would help us run off-the-shelf applications and expand the system evaluation to a larger set of applications.

References

[1] R. Arpaci, A. Dusseau, A. Vahdat, T. Anderson, and D. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of the 1995 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 267–278, May 1995.

[2] N. J. Boden, D. Cohen, R. E. Felderman A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

[3] D. Comer and J. Griffioen. A new design for distributed systems: The remote memory model. In *Proceedings of the Summer 1990 USENIX Conference*, pages 127–135, June 1990.

[4] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative Caching: Using remote client memory to improve file system performance. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation,*, pages 267–280, November 1994.

[5] M. de Prycker. *Asynchronous Transfer Mode: solution for broadband ISDN.* Ellis Horword, West Sussex England, 1992.

[6] T. Von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

[7] Schoinas et. al. Implementing Fine-Grain Distributed Shared Memory On Commodity SMP Workstations. Technical report, University of Wisconsin at Madison, Department of Computer Science, 1996.

[8] H. A. Jamrozik et.al. Reducing Network Latency Using subpages in a Global Memory Environment. In *Proceedings of the seventh ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 258–267, October 1996.

[9] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a Workstation cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 201–212, December 1995.

[10] E. W. Felten and J. Zahorjan. Issues in the implementation of a remote memory paging system. Technical Report 91-03-09, Department of Computer Science and Engineering, University of Washington, March 1991.

[11] M. J. Franklin, M. J. Carey, and M. Livny. Global memory management in client-server DBMS architectures. In *Proceedings of the 18th VLDB Conference*, pages 596–609, August 1992.

[12] P. J. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson, and B. L. Stumpf. The architecture of an integrated local network. *IEEE Journal on Selected Areas in Communications*, 1(5):842–857, November 1983.

[13] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95*, 1995.

[14] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, November 1996.

[15] The Spec92 Benchmark Suite, Release 1.1 , 1992.

[16] M. Talluri and M. D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *Proceedings of the 6th Int. Conf. on Arch. Support for Programming Languages and Operating Systems*, pages 171–182, October 1994.

[17] G. Voelker, H. Jamrozik, M. Vernon, H. Levy, and E. Lazowska. Managing Server Load in Global Memory Systems. In *Proceedings of the 1997 ACM Sigmetrics Conference on Performance Measurement, Modeling, and Evaluation*, pages 127–136, June 1997.